Software Model Checking with a Combination of Explicit Values and Predicates

Viktória Dorina Bajkai[1], Ákos Hajdu[1,2]
[1]Budapest University of Technology and Economics, Department of Measurement and Information Systems
[2]MTA-BME Lendület Cyber-Physical Systems Research Group
Email: hajdua@mit.bme.hu

***Abstract*—Formal verification techniques can both reveal bugs or prove their absence in programs with a sound mathematical basis. However, their high computational complexity often prevents their application on real-world software. Counterexample-guided abstraction refinement (CEGAR) aims to improve efficiency by automatically constructing and refining abstractions for the program. There are several existing abstract domains, such as explicit-values and predicates, but different abstract domains are suitable for different kinds of software. Therefore, product domains have also emerged, which combine different kinds of abstractions in a single algorithm. In this paper, we present a new variant of the CEGAR algorithm, which is a combination of explicit-value analysis and predicate abstraction. We perform an experiment with a wide range of software systems and we compare the results to the existing methods. Measurements show that our new algorithm can efficiently combine the advantages of the different domains.**

I. Introduction

Formal verification techniques (e.g. model checking) provide a sound mathematical basis to prove the correct operation of a program by exhaustively analyzing all possible states and transitions. This is also the downside of such methods, making them too expensive computationally. Counterexample-guided abstraction refinement (CEGAR) [1] is a widely used approach to overcome this limitation. CEGAR applies abstraction to hide certain details and to over-approximate the set of possible behaviors of the program. However, this does not only yield a smaller state space, but can also lead to spurious counterexamples. In such cases, the algorithm automatically refines the abstraction and repeats this process until a sufficient precision is found. CEGAR can work with different abstract domains, such as explicit values and predicates. The former works by tracking only a subset of the program variables, while the latter stores certain relationships and facts about them.

Product abstractions [2], [3], which combine multiple abstract domains also emerged, since different abstract domains turned out to be suitable for different kinds of software. In this paper we present a product abstraction algorithm, which combines explicit values [4] and predicate abstraction [5], exploiting the advantages of each domain. The key idea of our approach is that we always start with explicit values to avoid handling formulas, but switch to predicates if there are too many values for a variable.

Partially supported by Nemzeti Tehetség Program, Nemzet Fiatal Tehetségeiért Ösztöndíj 2018 (NTP-NFTÖ-18).

We implemented this algorithm in Theta [6], an open source verification framework, which already includes a generic CEGAR loop and some basic abstract domains. We evaluate our new approach and the existing methods on several programs from different problem domains, including PLC models form CERN [7] and C programs from the Competition on Software Verification (SV-Comp) [8]. Our results show that the new product abstraction algorithm successfully combines the advantages of two existing domains.

Related work: The dynamic precision adjustment approach [2] combines predicates and explicit values. The main difference is that it switches to predicates based on the whole state space, whereas we only consider the successors of a single state. Moreover, we allow successors to be enumerated if an expression cannot be evaluated, as opposed to the unknown values in dynamic precision adjustment.

Refinement selection [3] chooses between the explicit and predicate domains based on various metrics for the refinement quality. Our approach always tries the explicit domain first, but switches to predicates if there are too many different values.

II. Background

A. Control flow automata

In our work, we use *control flow automata* (CFA) [9] to model programs. A CFA is a tuple (V, L, l_0, E) where

- $V = \{v_1, v_2, \dots, v_n\}$ is a set of program *variables* with domains $D_1, D_2, \dots, D_n$,
- $L = \{l_1, l_2, \dots, l_k\}$ is a set of program *locations* representing the program counter,
- $l_0 \in L$ is the *initial location*, i.e., the entry point of the program,
- $E \subseteq L \times \mathit{Ops} \times L$ is a set of *directed edges* between locations, representing the operations that get executed when going from the source location to the target.

A *concrete state* $c = (l, d_1, \dots, d_n)$ of the CFA consists of a location $l \in L$ and a value $d_i \in D_i$ for each variable v_i from its domain. A *transition* $c \xrightarrow{op} c'$ exists between two states, if there is an edge $(l, op, l') \in E$ between their locations and the semantics of op matches the variables. Operations $op \in \mathit{Ops}$ can be *assumptions*, *assignments* or *havocs*. Assumptions are first order logic (FOL) [10] predicates denoted by $[\varphi]$, which must hold at the source state. Assignments are in the form of $v_i := \psi$, where ψ is a FOL expression with domain D_i that

updates the variable v_i in the target state. Havocs have the form *havoc* v_i, where v_i is assigned a non-deterministic value in the target state. A *concrete path* $c_1 \xrightarrow{op_1} c_2 \xrightarrow{op_2} \dots \xrightarrow{op_{n-1}} c_n$ is a sequence of concrete states and operations, where c_1 has the initial location l_0.

A *verification task* consists of a *CFA* and a dedicated *error location* $l_e \in L$. A CFA is *safe* if no concrete path exists to a state which contains the error location l_e.

As an example, consider the CFA in Figure 1a. It represents a program, which first examines whether its single variable x is not 1 and then if it is 1, leading to the error location l_e. Otherwise, the program ends in the final location l_f. It is clear that no concrete path can reach l_e, thus the CFA is safe.

B. Counterexample-guided abstraction refinement

In this section, we define the abstract CEGAR framework, which will be instantiated in the next paragraphs. CEGAR can work with different abstract *domains* [9]. An abstract domain is a structure $(S, \sqsubseteq, \Pi, T)$ where

- $S = \{s_1, s_2, \dots, s_n\}$ is a set of *abstract states*,
- $s \sqsubseteq s'$ is a *coverage relation*, which holds for two abstract states, if s' represents all the states that s does,
- Π is a set of *precisions*, which controls granularity of the abstraction,
- T is the *transfer function*, defining the successor relation between abstract states.

In this work we combine two different domains: explicit-value analysis [4] and predicate abstraction [5].

The first step of CEGAR is to build an *abstract reachability graph* (ARG) from the original model, with an initial, usually coarse *precision* $\pi \in \Pi$. ARG generation maintains a queue Q for the unprocessed states, starting with the initial abstract state $s_0 \in S$, corresponding to l_0. As long as the queue is not empty, it picks a state $s \in Q$ and checks if it can be covered with some already explored state s', i.e., $s \sqsubseteq s'$. If not, the successors of s are added to the queue for each operation *op* on the outgoing edges using the transfer function $T(s, op, \pi)$. Abstraction stops if the queue Q is empty or a state with the error location l_e is reached. Since the abstraction we use over-approximates the program, if we cannot reach l_e, the original program is also safe. Otherwise, an abstract counterexample exists, which is a path in the ARG leading to a state with l_e.

The second step is to examine the abstract counterexample. This is done by the *refiner* R, which checks whether the counterexample is feasible in the original program. If it is, the original program is unsafe. Otherwise, the counterexample is spurious and the abstraction is refined. The refiner R typically returns a new precision π' that should be joined to the previous $(\pi \cup \pi')$, so that the spurious counterexample is eliminated from the next iteration. The process repeats until there are no counterexamples or a feasible one is found.

C. Explicit-value analysis

Explicit-value analysis [4] works by tracking only a subset of the program variables. A precision $\pi_e \in \Pi_e$ defines the subset of the variables $\pi_e \subseteq V$, which are currently tracked. If a variable is not tracked, its value is represented by a special *top element* $\top$, which means that it can take any value from its domain. Abstract states $s_e = (l, d_1, \dots, d_n)$ in explicit-value analysis therefore, consist of a location l and values $d_i \in D_i \cup \{\top\}$ for each variable v_i. The coverage relation $s \sqsubseteq s'$ holds between two states, if their locations are equal and each value in s' is either $\top$ or the same as in s. The transfer function T_e works in the following way for a given state s_e, operation *op* and precision π_e. If *op* is an assumption and evaluates to true or cannot be evaluated (due to $\top$ values), a successor state is created where the value of the tracked variables will not change. If *op* is an assignment, the value of the assigned variable in the successor will be the result of evaluating the expression, or $\top$ if it cannot be evaluated or the assigned variable is not tracked. If *op* is a havoc, the value of the havocked variable becomes $\top$. Abstraction usually starts with an empty precision $\pi_e = \emptyset$ and refinement R_e is performed by iteratively extending the precision π_e with additional variables. The new variables to be tracked can be inferred by different *interpolation* techniques [4], [11].

As an example, consider the ARG in Figure 1b (for the CFA in Figure 1a) created with explicit-value analysis. The value of x is not known initially and also remains $\top$ after $[x \neq 1]$. However, after $[x = 1]$ we know that x is 1.

D. Predicate abstraction

In predicate abstraction [5], the concrete values of variables are not tracked explicitly. Instead, certain facts and relationships are tracked through a set of FOL formulas over V, called the *predicates*. The precision $\pi_p \in \Pi_p$ is therefore, a set of predicates. Abstract states $s_p = (l, p_1, \dots, p_k)$ contain the location and the ponated or negated version of the predicates p_i in π_p. It is also possible that a predicate does not occur in an abstract state, if it can both hold or not. The coverage relation $s \sqsubseteq s'$ holds between two states if their locations are equal and the predicates of s imply the predicates of s'. The transfer function T_p works in the following way for a given state s_p, operation *op* and precision π_p. If *op* is an assumption, we check whether the conjunction of the predicates of the source state and the assumption is satisfiable. If yes, a successor state is created, including the predicates from π_p (or their negated form) that are implied by the source state and the assumption. If *op* is an assignment, we create a successor that includes the predicates or their negated form that are implied by the source state and the assignment. If the operation is a havoc, a successor state is created, where predicates including the havocked variable are excluded. Abstraction usually starts with an empty precision $\pi_p = \emptyset$ and refinement R_p is performed by iteratively extending the precision π_p with additional predicates. Similarly to explicit values, the new predicates can be inferred by interpolation [11].

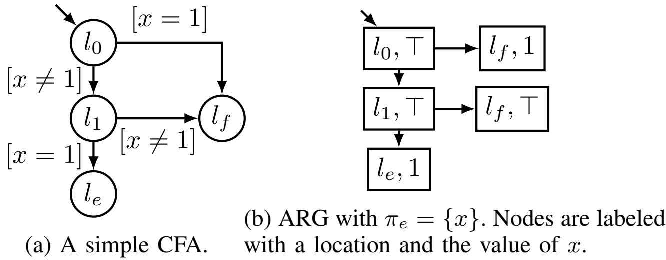
As an example, consider the ARG in Figure 2b (for the CFA in Figure 1a) with tracking the predicate $x = 1$. The assumption $[x \neq 1]$ ensures that l_1 is labeled with the negation of the predicate, and thus the error location will not be reachable in the ARG.

E. Product abstraction

Product abstractions [2], [3] combine different abstract domains. In our work we combine explicit-value analysis and predicate abstraction by tracking both explicit values and predicates simultaneously. Therefore the precision is $\Pi = \Pi_e \times \Pi_p$ and an abstract state $s = (l, d_1, \ldots, d_n, p_1, \ldots, p_k)$ consists of a location, values and predicates. A state s is covered by another state s' if their locations are equal and both components are covered. The transfer function T gets a product state $s = (l, s_e, s_s)$, an operation op, a precision $\pi = (\pi_e, \pi_p)$ and calculates $T_e((l, s_e), op, \pi_e) \times T_p((l, s_p), op, \pi_p)$, that is the Cartesian product of the successor explicit and predicate states. Abstraction usually starts with an empty precision $\pi = (\emptyset, \emptyset)$ for both components. In product abstraction, the main decision regarding the new precision should be made during refinement R. The component refiners R_e and R_p will return new variables π'_e and predicates π'_p to be tracked and the algorithm has to decide which of them to use. In the next section we propose a new strategy to choose between explicit values and predicates.

III. Product Strategy with Limited Enumeration

The key idea of our approach is that we always extend the set of explicitly tracked values (π_e) first, since handling predicate formulas is more expensive computationally (e.g., checking implications). However, a downside of explicit-value analysis is that some problems are not decidable due to expressions with unknown values ($\top$) that cannot be evaluated.



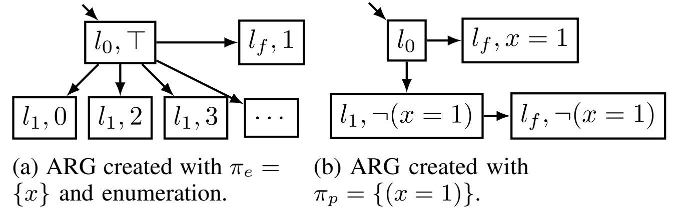
(a) A simple CFA. (b) ARG with $\pi_e = \{x\}$. Nodes are labeled with a location and the value of x.

Fig. 1: Example CFA and ARG with explicit-value analysis.

Recall the example CFA in Figure 1a, which first checks if $x \neq 1$ and then if $x = 1$ (which obviously cannot be possible). If no variables are tracked initially, the error location is trivially reachable and the refiner extends the precision with the only variable x. Figure 1b shows the corresponding ARG with $\pi_e = \{x\}$. Since x is not initialized, the initial state is $(l_0, \top)$. Then we check the condition $x \neq 1$. Since x is unknown, the condition can both hold and not. If it does not hold, the program terminates in the final location $(l_f, 1)$. However, if it holds we proceed to l_1, where x is still $\top$, since we cannot represent the fact that $x \neq 1$ in explicit value analysis. Then we check the condition $[x = 1]$, which can again hold or not, due to x being unknown. This way, the program can still reach the error location $(l_e, 1)$, which is a spurious counterexample. Since there are no more variables to be tracked, the program cannot be verified (with explicit-value analysis).

To address such limitations, we propose a modified version of the explicit transfer function T_e, where we start to list all possible values instead of using a $\top$ value if an expression cannot be evaluated. This can already solve the problem for some cases, e.g., an assumption $[0 < x < 5]$ would yield 4 successors with 4 values for x (instead of a single successor where $x = \top$). However, this can also easily lead to state space explosion. As an example, consider the CFA in Figure 1a again, where x was already added to the set of explicitly tracked variables as previously ($\pi_e = \{x\}$). The corresponding ARG for this precision can be seen in Figure 2a. The program starts at state $(l_0, \top)$, from where it can go in two different directions. Taking the assumption $[x = 1]$, it arrives at state $(l_f, 1)$ since $x = 1$ is the only possible value satisfying the formula. Otherwise, the program moves to l_1, where it starts to list the possible values for $[x \neq 1]$, which obviously leads to a state space explosion.

To overcome this issue, we also introduce a limit k. During enumeration in T_e, we count the different values for each explicitly tracked variable (in π_e). If the number of different values of a variable v_i in the successor states exceeds k, we remove it from the explicit precision ($\pi_e := \pi_e \setminus \{v_i\}$) and also mark it with a flag, so that the refiner will not include it again. Since the precision changed, we restart the enumeration in T_e, but now with a new precision π_e. We repeat this process until there are no more successor states to be enumerated and no variable was excluded.



(a) ARG created with $\pi_e = \{x\}$ and enumeration. (b) ARG created with $\pi_p = \{(x = 1)\}$.

Fig. 2: Examples for the new product abstraction algorithm.

During refinement R we use both the explicit refiner R_e and the predicate refiner R_p to obtain new variables π'_e and predicates π'_p to be tracked. We loop through the new variables $v_i \in \pi'_e$ and check if they are marked with the flag. If they are, we do not include them, but rather extend the predicate precision π_p with those predicates in π'_p that contain v_i. Otherwise, we extend the explicit precision π_e with v_i.

Recall the ARG in Figure 2a again. Our new transfer function T_e stops enumerating values for x after a finite k, removes x from the set of explicitly tracked variables π_e and restarts the enumeration. However, now x is not tracked anymore, so we trivially reach the error location (similarly to Figure 1b but we reach $(l_e, \top)$, since x is not tracked). The product refiner R will not add x again, since it is flagged. Instead, it uses R_p to add some predicate, e.g., $x = 1$ to the precision π_p. Figure 2b shows the ARG created with the new precision. From l_0, the program can arrive to the final location $(l_f, x = 1)$ where the predicate is true, or move to $(l_1, \neg(x = 1))$ where the negation of the predicate holds.

At this point, the predicates keep track that $x \neq 1$, so the algorithm can only proceed to $(l_f, \neg(x = 1))$. Since there are no more states to explore and the algorithm did not reach the error location, the program is safe. For this example, product abstraction first used x, then discarded it and used a predicate instead. However, in general it is possible that some variables remain explicitly tracked, while others have predicates.

IV. Evaluation

We implemented the algorithm in the open source Theta framework [6], which already includes the explicit and predicate domains, and the Z3 SMT solver [12]. We implemented a modified transfer function for the explicit domain and a refinement procedure for product abstraction. We ran measurements on 90 PLC (programmable logic controller) programs from CERN [7] and 340 C programs from the Competition on Software Verification (SV-Comp) [8], containing large event-driven systems (eca), small locking mechanisms (locks) and large server-client systems (ssh). We evaluated these programs with eight different configurations: explicit-value analysis (EXPL), predicate abstraction (PRED) and our new product strategy (PROD) with six different limits ($k = 1, 2, 4, 8, 16, 32$). We ran the measurements on a 64 bit Ubuntu 16.04 OS using the RunExec tool from the BenchExec suite [13], which ensures highly accurate results. We enforced a time limit of 180 seconds and a memory limit of 4 GB.

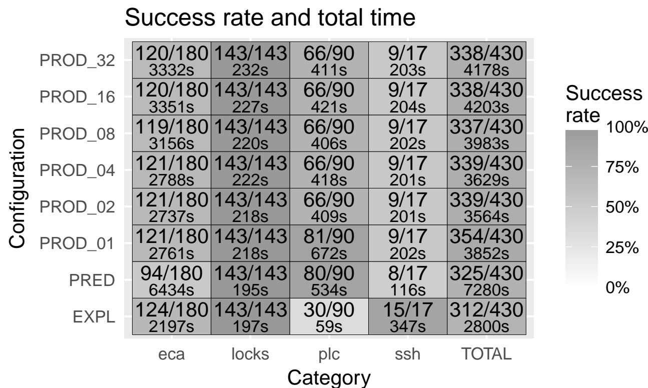


Fig. 3: Heatmap of the results for each configuration in each category. Cells include the number of verified models among the total and the required time for the successful executions.

The heatmap in Figure 3 shows the results of our evaluation. Rows correspond to the configurations, while columns represent categories. The last column is a summary of all categories. Each cell contains the number of successfully verified models and the total number in that category. We also included the execution time (in seconds) required for the successful runs. We can see, that the product abstraction strategies have better overall performance than explicit values and predicates. Furthermore, product abstraction with $k = 1$ has the best overall performance, verifying a total number of 354 models.

In category plc, PRED is successful but EXPL is not, and the eca category is the other way around. However, the PROD strategies (especially with $k = 1$) provide a good performance in both categories, combining the advantages of the two base domains. The locks category was easy for each configuration. The ssh category is interesting, because EXPL performs well, but the PROD strategies are closer to PRED with a rather poor result. This would require further investigation.

In general, the overall results confirm that our product abstraction strategy can successfully combine the strengths of explicit-value analysis and predicate abstraction.

V. Conclusions

In our paper we investigated CEGAR-based software model checking and presented a new product abstraction strategy, which combines explicit values, enumeration and predicate abstraction. We implemented the new algorithm in the Theta verification framework, ran measurements on various input programs and compared it to existing domains. Our experiment shows that the new algorithm can successfully combine the advantages of explicit-value analysis and predicate abstraction, yielding a more efficient model checking strategy.

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[2] D. Beyer, T. A. Henzinger, and G. Theoduloz, "Program analysis with dynamic precision adjustment," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 29–38.

[3] D. Beyer, S. Löwe, and P. Wendler, "Refinement selection," in *Model Checking Software*, ser. LNCS. Springer, 2015, vol. 9232, pp. 20–38.

[4] D. Beyer and S. Löwe, "Explicit-state software model checking based on CEGAR and interpolation," in *Fundamental Approaches to Software Engineering*, ser. LNCS. Springer, 2013, vol. 7793, pp. 146–162.

[5] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, ser. LNCS. Springer, 1997, vol. 1254, pp. 72–83.

[6] T. Tóth, A. Hajdu, A. Vörös, Z. Micskei, and I. Majzik, "Theta: a framework for abstraction refinement-based model checking," in *Proc. 17th Conf. on Formal Methods in Computer-Aided Design*. FMCAD inc., 2017, pp. 176–179.

[7] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Trans. on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[8] D. Beyer, "Software verification with validation of results," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, 2017, vol. 10206, pp. 331–349.

[9] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Computer Aided Verification*, ser. LNCS. Springer, 2007, vol. 4590, pp. 504–518.

[10] A. R. Bradley and Z. Manna, *The calculus of computation: Decision procedures with applications to verification*. Springer, 2007.

[11] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik, "A configurable CEGAR framework with interpolation-based refinements," in *Formal Techniques for Distributed Objects, Components and Systems*, ser. LNCS. Springer, 2016, vol. 9688, pp. 158–174.

[12] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 4963, pp. 337–340.

[13] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *International Journal on Software Tools for Technology Transfer*, 2017, online first.